

Doc. no. P2687R0

Date: 22022-10-15

Project: Programming Language C++

Audience: EWG

Reply to: Bjarne Stroustrup (Bjarne@stroustrup.com)

Design Alternatives for Type-and-Resource Safe C++

Bjarne Stroustrup (Columbia University)

Gabriel Dos Reis (Microsoft)

Abstract

We discuss a set of alternatives for achieving type-and-resource safe programming in the context of C++. After discussing alternatives, we propose a set of minor additions to ISO C++ to ease static analysis to provide guarantees for code obeying a variety of coding rules.

The alternatives are examined under the constraints imposed by existing standards, tool chains, coding styles, needs for stability, needs for flexibility, needs for performance, needs for a variety of kinds of safety, and the need to interoperate with code written in other languages in a wide variety of application areas supporting many billions of lines of C++ code developed by several millions of C++ programmers.

1. The problem

It is easy to break the C++ type system: misuse of unions, dangling pointers, range errors, misuse of casts, etc. Obviously, such breakage is usually not deliberate and can to some extent be avoided (witness the world's massive largely successful use of C++ in applications and infrastructure). However, the possibility of such breakage leads to extra work for developers and occasionally to bad errors and security violations.

In our opinion, this needs to be addressed. Saying "but you can write good C++ code" is not enough because many developers don't. It is very hard to ensure absence of errors merely through "being careful." Also, "being careful" is ill-defined and subject to a variety of opinions. We need guarantees, preferably static, against violations. Where guarantees require run-time checks (e.g., range checking), we must guarantee that the run-time checks occur.

In our opinion, this is not a minor problem that can be ignored. Not addressing it could easily lead to disuse of C++ in key areas where it would otherwise be the best choice of language.

Appendix A is a collection of unsafe C++ constructions and how we propose to prevent them. Each of these examples could possibly lead to a violation for some values and is accompanied by safe alternatives.

1.1 Type-and-resource safety

There are several notions of what constitutes safety violations

- *Logic errors*: perfectly legal constructs that don't reflect the programmer's intent, such as using `<` where a `<=` or a `>` was intended.
- *Resource leaks*: failing to delete resources (e.g., memory, file handles, and locks) potentially leading to the program grinding to a halt because of lack of available resources.
- *Concurrency errors*: failing to correctly take current activities into account leading to (typically) obscure problems (such as data races and deadlocks).
- *Memory corruption*: for example, through the result of a range error or by accessing and memory through a pointer to an object that no longer exists thereby changing a different object.
- *Type errors*: for example, using the result of an inappropriate cast or accessing a union through a member different from the one through which it was written.
- *Overflows and unanticipated conversions*: For example, an unanticipated wraparound of an unsigned integer loop variable or a narrowing conversion.
- *Timing errors*: for example, delivering a result in 1.2ms to a device supposedly responding to an external event in 1ms.
- *Termination errors*: a library that terminates in case of "unanticipated conditions" being part of a program that is not allowed to unconditionally terminate.

Our approach is based on the C++ Core Guidelines [CG] and the idea of Semantically Enhanced Language Libraries which dates back to the early 2000s [SELL]. This work started long before the current "emergency" and can in principle be applied to all of these kinds of errors. However, this specific proposal addresses "just" the errors related to types and resources. Some, but not all, concurrency errors are covered by considering threads as containers (§A.5. Data Races).

We address two related issues: type safety and resource safety:

- By *Type safety*, we mean that every object is used only according to the type with which it was defined. That implies proper initialization, the avoidance of type punning, and preventing access through pointers of inappropriate types. It also implies that access to an object after the end of its lifetime must be prevented.
- By *Resource safety*, we mean that every object that represents ("owns") a resource releases its resource at the end of the owner's life and that no access is made to an object after the end of its lifetime. That requires that the use of such owners is type safe and that their types have a suitable set of operations, such as copy, move, and destructor to achieve that. An owner object allocated on the free store and never deleted is considered to live forever.

Obviously, resource safety can be achieved only if type safety exists. Conversely, type safety can be achieved only if resources are not leaked so that programs don't fail due to resource exhaustion. In particular, to guarantee type-and-resource safety, we need to guarantee that a pointer never points to a destroyed object, so it is a generalization of memory safety.

1.2 Guarantees

Our ideal is static guarantees. That is, guarantees that certain classes of errors cannot occur in a running program. This can be ensured by compilers, other kinds of static analyzers, or combinations thereof.

However, some kinds of errors, such as range errors and dereferences through a pointer holding a nullptr, cannot be detected at compile time. In such cases, we need guarantees that a run-time check is executed.

Finally, through hardware failures or compiler bugs, “impossible” errors can happen. We do not address such errors directly but recommend that critical software occasionally perform sanity checks aimed at detecting these particularly rare and troublesome kinds of errors. Such sanity checks can also be useful for detecting unanticipated software errors.

2. Alternative solutions

2.1. A safe subset

From the earliest days of C++, there have been a stream of suggestions to define a safe subset of C++. The obvious idea is to define a compiler option that prohibits unsafe operations. Unfortunately, since a compiler doesn’t have a global view of a program, that would have to include bans of most:

- Casts
- Unions
- Pointer arithmetic
- Arrays
- Explicit lifetime management of objects (using placement new and explicit destruction)

And probably more. In other words, the language would no longer be a viable C, let alone C++. The deeper problem is that we need low-level (potentially unsafe) operations to implement our safe abstractions efficiently. All languages dealing directly with memory and also aiming at safety must face that. Usually, the solution is some form of trusted code segments or simply a foreign language call interface, leaving the dirty work to C or C++.

We adopt aspects of the trusted code idea (§5.6, §7.4).

2.2. A safe dialect

A solution to the safe-subset dilemma is to define a dialect with language extensions that offers sufficiently safe facilities to render use of unsafe facilities unnecessary, or at least rare and isolated in trusted code segments. This too has been tried repeatedly. The safe facilities can be core-language facilities and/or libraries known to the compiler. These extensions tend not to be ISO standard. The problem then becomes that the design of the safe dialect becomes the design of a superset to offer the safe features plus the design of interfaces between the novel safe features and parts of the old language deemed safe (often requiring annotations of pointers and references). In the “safe dialects of C and C++” we know of, exclusive use of the non-ISO-standard “safe” extensions has been required for essentially all serious programming. The extensions’ containers (e.g., vectors, lists, or hash tables) and all references to their elements cannot use unannotated pointers. What’s left is not anything like C or C++, but a new language plus a strained interface to C or C++.

We adopt aspects of the safe-dialect idea (§4).

2.3. A new somewhat compatible language

Every modern language needs to be able to interoperate with C or C++. If the interface between the new language and C++ is sufficiently elegant, efficient, and familiar to C++ programmers a transition to the new

language seems plausible. The old code is preserved behind an interface and possibly separately compiled whereas the shiny new code is unconstrained by C and C++ compatibility concerns. That, at least, is the fundamental idea, but the details can be messy when we try to scale to larger, more demanding systems.

We must now have two tool chains dealing with front-end needs, such as IDEs, compiler front-ends, high-level static analyzers, database interfaces, system interfaces, graphics interfaces, etc. C++ will not magically disappear and the two tool chains will have to be maintained in parallel for a decade or more. Many developers will have to become familiar with both languages and tool chains. The complexity of the development and teaching will increase.

The efficiency and flexibility of the new language must be considered suspect for years as it will not have been applied to the wide range of application areas and systems where C++ is in use. Using a C or C++ implementation substrate, such as LLVM or Linux with suitable modifications, helps but can also hamper the exploitation of guarantees offered by the new language. For example, C++ optimizers were not able to fully utilize C++'s const rules because the optimizers were designed for C, and C's definition of const was different from C++'s. Also, close inter-operation with C and C++ implies the need to conform to the notoriously inflexible platform ABIs guarding their backwards binary compatibility.

Finally, when considering safety guarantees, essentially every C and C++ library used, and C and C++ function called must be considered trusted, thus making the guarantees offered by the new language conditional upon the correctness of potentially millions of lines of traditional, unverified C++.

We address the thorny issue of combining code with differing guarantees in §7.

2.4. A modernized C++

Abandon compatibility with C and older ISO C++, and ban unsafe features, such as arrays, unions, and pointer arithmetic. This approach is a more ambitious variant of the idea of a "Safe C++" dialect in that it aims to be C++, leaving the older version as non-conforming dialects.

There are billions of lines of C++ deployed. Thus, changing C++ significantly in incompatible ways will affect millions of programmers and potentially billions of users. This is not feasible. We need to enable and ease a gradual transition to more modern styles together with offering guarantees for critical code. Please note that ABIs are controlled by platform providers and rather than by the standards committee and that an unknown, but huge, set of applications run on a platform, making ABIs very hard to change. In general, a significant ABI change requires a global update, and nobody knows all the users.

This implies that the older C++ features will not go away, but even if banned will be preserved (e.g., through compatibility switches) and the compiler and library suppliers will have to maintain them "forever" together with novel features.

We are all for modernizing C++ by adding significant improvements (e.g., functional-programming-style pattern matching and static reflection) **but see a strong need to do so compatibly**. Unless done extremely carefully, the "modernized C++" would suffer from the problems encountered with "Safe C" languages: the inability to interoperate sufficiently smoothly with older C++ constructs to avoid needing to become a self-sufficient subset of a much-enlarged C++ language.

2.5. Patching

It is always popular to do the minimal patch to the current system to make a problem disappear. For example, adding a checked vector for people to use exclusively. Unfortunately, such patches tend to be incomplete because they leave the fundamental complexity of the code alone. Even if it works, patching leaves behind the old-style convoluted, error-prone, and hard-to-maintain code. To move forward, we must find ways to support a move towards simpler and safer programming techniques. A reduction of code complexity supported by compile-time guarantees is necessary to keep the cost and complexity of run-time checks comparable to well-written existing C++ code [SELL] [CG].

We adopt a principled variant of patching by implicitly adding run-time checks where guarantees are required but static guarantees are impossible (§6).

2.6. Guidelines

Leave ISO C++ alone except for adding a few annotations and library components, but limit what can be written “safely” though a set of rules enforced by static analysis. The syntax and semantics of every construct are defined by the ISO C++ standard. The annotations act only to impose restrictions and/or help static analysis.

This is our proposed approach. To a much larger extent than other alternatives, it supports gradual adoption, partial adoption, and alternative views of what constitutes safety. Also, static analysis is a good first step to code transformation, which in our opinion is a necessary step to modernizing older code bases. We did some experiments with that 10+ years ago.

3 The C++ Guidelines

The approach of guidelines supported by libraries and local static analysis was pioneered by The C++ Core Guidelines. The underlying philosophy and the current set of rules for the C++ Core Guidelines are available on GitHub [CG]. Partial implementations of static analysis support can be tried in Microsoft Visual Studio’s analyzer, JetBrains’ ReShaper, and Clang Tidy. At the time of writing, the VS checker is the most complete.

The CG is based on the observation that restrictions on language use, libraries, and static analysis each can increase the degree of safety, but each on its own imposes restrictions, compile-time costs, run-time costs, or convoluted code that makes them unacceptable to many. On the other hand, **a judicious combination of all three within a principled framework is viable**. Please read the CG Introduction for a more thorough explanation.

The set of rules reduces the complexity of the code to something that is tractable by mostly local static analysis supported by just a few run-time checks. Without rules limiting the complexity of code, static analysis would be impossible in many cases and not scale to realistically sized programs. Some of the options and rule checks could and should be performed by a compiler, but at the current state of compilers, we consider doing a complete check of arbitrary rules too much overhead in many development/debugging environments. The alternative is having a static analyzer in the tool chain and offering guarantees that it is always invoked before deploying critical code.

The actual rules from the C++ Core Guidelines are not described here. For those, see [GC].

4. Proposed guidelines support

The current C++ Core Guidelines rests on three pillars

- Static analysis, using an AST-based type analysis plus a control flow analyzer and relying on local analysis to allow the analysis to scale and limit its cost.
- Library components to provide alternatives to known error-prone constructs such as casts and subscribing naked pointers.
- Rules guiding users to improved coding techniques and – essentially – reducing the complexity of code to the point where static analysis is feasible.

Some minor support in the standard would be most helpful – and likely essential – for this approach to be applicable everywhere. In particular, it would be possible to dramatically limit or even eliminate the GSL (The Guidelines Support Library) in favor of the standard library.

Please note that we do not offer a painless transition. Significant progress and meaningful guarantees are possible only through a simplification of application code. Critical code will have to be simplified (e.g., changing pointer use to span use) or rearchitected. New code can be written using modern techniques, modern ISO C++ language facilities, and libraries with modern interfaces (e.g., type-and-resource-safe interfaces).

4.1. Precedence

The support proposed here is based on five years’ experience with the C++ Core Guidelines and observations from languages and libraries deemed safe. In particular, all such languages and libraries that allow direct access to memory have a notion of trusted code or a foreign-language call interface to allow potentially “unsafe” operations to be done in C, C++, or assembler.

The majority of examples of problematic constructs that can be caught by a static analyzer are, in fact, already caught by the VS analyzer. Thus, though this paper is focused on ideas, it is not just speculation.

Complete type-and-resource safety has been the ideal for C++ from very early on (the early 1980) [DnE]. Obviously, the path towards that has been long because of constraints of developer culture, performance needs, needs for flexibility, compatibility, etc. The basic model for safety in the Core Guidelines is outlined in [Stroustrup’15].

4.2. Overview

The high-level Ideas behind the detailed proposals are

- The meaning of all constructs is defined by the ISO C++ standard
- The most fundamental guarantee offered is complete type-and-resource safety
- Gradual conversion from older code to modern code offering guarantees is supported
- Ownership (that is, the obligation to delete/destroy) constitutes a DAG
- A pointer is the **nullptr** pointer, or is valid
- A pointer (outside the implementation of abstractions) points to a single object
- There is a way to enforce **nullptr** checking and range checking
- Subscripting is done on abstractions such as span and vector
- The set of guarantees is open
- A set of fundamental guarantees are standard

- There are rules for composing code fragments supporting different guarantees.
- The set of guarantees assumed by and provided by a unit of code is stated in the code.

Guarantees are established by a combination of enforceable coding rules, foundational libraries, and mostly local static analysis.

Because of time pressure, these proposals are not completely fleshed out, but what's presented should give a good idea of the general direction and many details.

5. Static analysis support

Our ideal is to catch as many problems as possible before a program starts executing. That way we avoid potentially costly (and possibly repetitive) run-time checks, simplifies the job for the optimizer, and saves us from writing error handlers.

For that, we need type-based analysis (an AST representation) and control-flow analysis.

5.1. Initialization

It is easy for a static analyzer to guarantee all objects are initialized. The only serious problem is that guaranteeing the initialization of large buffers can be costly. We may need a **[[uninitialized]]** annotation or an **uninitialized** pseudo initializer to indicate that an object is deliberately not initialized. Such code would obviously fall into the unverified category and be flagged for extra attention in code reviews or banned in some very critical code.

5.2. Ownership

The CG recommendation is to rely on ownership abstractions (such as **vector** and **unique_ptr**) but the implementation of such container abstraction and in interfaces to C-style code, we need a lower-level notion of ownership. The GSL offers a simple aid to allow a user to tell a static analyzer that a pointer points to an object that needs to be deleted:

```
template<typename T> using owner = T;

int* f(owner<int*> p)
{
    X* q = new X{ /* initializers */ };
    // use *p
    // error: no delete p;
    // error: no delete q;
    return p;    // error: p returned without owner indicator
    return q;    // error: q returned without owner indicator
} // error
```

The static analyzer can catch such problems, including ensuring proper transfer of owners between functions.

An alternative would be to use an annotation, e.g., **[[owner]]**.

5.3. Dangling pointers

Dangling pointers can and must be avoided. For example, writing through a dangling pointer can corrupt an unrelated object. We get a dangling pointer by retaining a pointer to a destroyed object (on the free store, on the stack, or in a use-defined pool) and then using it for reading or writing.

There are three basic cases:

- Use a pointer after the object to which it refers has been deleted in that scope.
- Let a pointer escape from the scope of the object to which it refers.
- Retain a pointer to an object after that object has been destroyed (“invalidation”)

Note that when we talk about pointers, we mean every construct that can refer to an object.

5.3.1. Local deletion

A free-store object that is constructed and destroyed in the same scope shouldn’t exist, but thanks to overuse of **new** (rather than local objects and smart pointers) such cases are not uncommon.

```
void f(int x)
{
    int* p = new int{7};
    int* q = new int{8};
    delete q;
    *q = 9;           // likely disaster (but we won't get here)
    if (x)
        delete p;
    *p = 10;         // likely disaster (but we won't get here)
}
```

There are two cases for a static analyzer:

- A pointed-to object is unconditionally deleted and then used. That’s easily caught.
- A pointed-to object is conditionally deleted (deleted on some path and not another) and then used. That cannot be caught by a static analyzer. It must be rejected.

Using a **unique_ptr**, or better still a local variable, is typically the solution to such cases.

5.3.2. Escaping pointer

The basic strategy is not to let any pointer escape from the scope of the object to which it refers. For example:

```
void f()
{
    int* p = nullptr;
    {
        int x = 7;
        p = &x;
        // ...
    }
    *p = 9; // likely disaster (but we won't get here)
```

```

}

```

Local static analysis can catch such problems and also reject examples where the escape is conditional.

The ownership rules can catch versions of this problem that are not local to a function.

```

void use(int* p)      // p is valid because it must be valid in the calling scope
{
    // ...
    delete p;        // error: p is not an owner
}

void f(int*p)
{
    use(p);
    *p = 7;         // likely disaster (but we won't get here)
}

```

5.3.3. Multiple pointers

Aliases pose a challenge to static analysis. This is one of the key cases where restrictive rules are needed to reduce complexity of analysis. Consider:

```

void f()
{
    auto p = new int(7);
    auto q = p;      // make an alias
    delete p;
    *q = 9;        // likely disaster (but we won't get here)
}

```

The static analyzer must keep track of such aliases. Keeping functions short is key to helping the analyzer. Limiting the number of aliases also helps a lot.

Unfortunately, smart pointers don't help much because programmers (unnecessarily) extract raw pointers from them:

```

void g()
{
    auto p = make_unique<int>(7);
    auto q = p.get();    // make an alias
    p = nullptr;
    *q = 9;           // likely disaster (but we won't get here)
}

```

The `get()` extracts a pointer from the `unique_ptr` container and the assignment leads to invalidation (§5.3.4).

5.3.4 Invalidation

This leaves the examples usually referred to as “invalidating.” We define “invalidated” as “when a pointer points to an element of a container that may have reallocated or deleted its elements.” For example:

```
void f(vector<int>& vi)
{
    vi.push_back(9);    // may relocate vi's elements
}

void g()
{
    vector<int> vi { 1,2 };
    auto p = vi.begin(); // point to first element of vi
    f(vi);
    *p = 7;             // likely disaster (but we won't get here)
}
```

For a static safety guarantee, we need to reject returns that can lead to a violation. Thus, this **f()** must be considered invalidating and **g()** must be rejected.

The CG offers a useful rule for avoiding invalidation:

- [Con.2: By default, make member functions const](#)

For this rule to be comprehensive, we must define “container” to include any object that directly or indirectly could contain a pointer:

- Classes with pointer members
- Lambdas (they are classes, and remember capture-by-reference)
- **Jthreads** (they are classes in a defined scope)
- **unique_ptrs** and **shared_ptrs** (they are classes with a single logical pointer member)
- Pointers to pointers
- References to pointers
- Arrays of pointers

Here, “pointer” can be anything that refers to an object (e.g., a **unique_ptr**). A “pointer member” of one of these can lead to an invalidation problem only if

- it is possible to point to an element pointed to by such a “pointer member”
- the value of the pointer member can be changed

For example:

```
int x = 7;
int* p = &x;
int** pp = &p;
cout << **pp;    // 7
p = nullptr;
cout << **pp;    // likely disaster (but we won't get here)
```

The VS static analyzer considers all functions that take a container by non-**const** reference invalidating and all functions that take a container by **const** reference non-invalidating.

- The latter is reasonable because we can statically verify that a **const** argument isn't mutated by the function. Con.2 also recommends "don't cast away **const**" and the type safety profile requires that. This is necessary for complete static safety.
- The former is probably too conservative: Containers often have non-**const** operations that modify the value of a container object without its structure, e.g., **vector::swap()** and **vector::operator[]()**.

We need a **[[not_invalidating]]** to annotate functions that potentially could invalidate, but don't. Considering non-**const** functions invalidating is a good and safe default. Therefore, the annotation should be used only to add to that; that is, we need **[[not_invalidating]]** rather than **[[invalidating]]**. A **[[not_invalidating]]** annotation can (and should) be statically verified by examining the definition of a supposedly **[[not_invalidating]]** function.

5.4. Memory pools

The discussion in §5.3 assumes that objects are static, local (automatic), or on the free store (heap, dynamic memory) managed by **new** and **delete**. However, user-defined memory management in various forms is essential in many application areas, and fundamental in the C++ standard library.

By a memory pool, we mean a section of memory in which an object can be stored. In principle, a pointer to an object in a memory pool can be handled in a similar manner to that of a pointer to an object created by **new**. However, C++ lacks a standard "pool" abstraction. Instead, there are thousands of variations of the idea, seriously complicating the task of static-analyzer writers. To avoid dangling pointers to its stored objects, a pool can apply one of alternative strategies:

- Disallow objects to be deleted or relocated
- Disallow pointers to objects to escape
- Invalidate all pointers to objects if a potentially deleting or relocating operation is invoked

std::vector with subscripting and **resize()** is a typical example of a pool that requires special attention and is dealt with through invalidation (the third alternative) enforced by static analysis. If a non-**const** function is invoked on a vector, all pointers to its elements are considered invalid and may not be used. This is ensured through static analysis. This is a conservative, but safe, strategy that can be applied to every pool. To enable a non-**const** function (e.g., **vector::operator[]()**) to be considered not invalidating, we need an **[[not_invalidate]]** annotation. That annotation can be validated by static analysis (§5.4).

5.6. Linked data structures

RAII essentially creates a tree of objects representing ownership and the order of creation and destruction is "first created, last destroyed." This leaves circular ownership structures as a nasty problem. This simplest, most obvious, and often most efficient solution is to avoid circular ownership structures.

However, some people like circular structures and some applications (such as general graphs) naturally give rise to circular structures. Furthermore, it is hard (impossible?) to statically ban circular ownership structures without banning some of the most popular linked structures.

Even a simple singly linked list can give rise to a circular structure:

```

template<class Value>
struct Link { Value val; Link* next; };

using Lnk = Link<int>;

Lnk* lnk1 = new Lnk{1,nullptr};
Lnk* lnk2 = new Lnk{2,lnk1};
lnk1.next = lnk2;           // circular list

```

Cleaning up that code using `unique_ptr` we get:

```

template<class Value>
struct Link { Value val; unique_ptr<Link> next; };

using Lnk = Link<int>;

auto lnk1 = make_unique<Lnk>(1,nullptr);
auto lnk2 = make_unique<Lnk>(2,lnk1);
lnk1.next = lnk2;

```

This more elegantly creates a leaked cycle.

Static analysis cannot in general detect the creation of such cycles but static analysis of instantiated types can easily catch the possibility of cycles. That is, we can reject structs, such as `Link`, that are potentially self-referential.

Having – of necessity – rejected such popular potential cyclic structures in programs that require strong safety guarantees, how can we use such structures where needed? The traditional solution is to declare the implementation of lists, trees, graphs, etc. “trusted.” That’s unsatisfactory because that leaves much tricky code unverified. Alternatively, we could require a garbage collector. That solution has other major problems in the context of C++. Among the reasons not to rely on classical garbage collection is that GC doesn’t handle non-memory resources, is non-local, doesn’t minimize resource retention (objects may live “forever”), and requires significant run-time support.

One way to eliminate the possibility of circular ownership structures is to separate ownership from access. That is essentially what the standard-library containers do by always holding copies of inserted elements. From the perspective of resource safety, values inserted using `emplace` functions are not fundamentally different from copies. Cycles in non-owning pointers are not a problem as they cannot be used to `delete` (§5.3).

Separating ownership from access, the binary tree could look something like this:

```

struct Tree_node2 {
    Value val;
    Tree_node2* left;
    Tree_node2* right;
};

struct Tree2 {

```

```

    vector<unique_ptr<Tree_node2>> nodes;
    Tree_node2* head;
    // ...
};

```

Where each node in the tree has its address added to **nodes**. Code adding nodes would transfer ownership to **nodes** or let **Tree2** functions create nodes.

Alternatively (and usually more efficiently in time and space), the nodes could all be allocated in a pool:

```

struct Tree_node3 {
    Value val;
    Tree_node3* left;
    Tree_node3* right;
};

struct Tree3 {
    Pool<Tree_node3> nodes;
    Tree_node3* head;
    // ...
};

```

The key here is to separate ownership from access. Like for STL containers, a container that allows deletion of elements must have a notion of invalidation (§5.3.4). Containers without deletion can be very useful and simplify analysis. A standard-library **Pool** would be most helpful.

How can we guarantee that the ownership structure of components such as the trees above are correctly specified and implemented? We don't see how that could be done statically for arbitrary data structures. That is, we haven't been able to think of a simple and efficient set of rules for limiting implementations to what can be verified to be safe while allowing all linked structures that developers are likely to reasonably want.

There are no CG rules covering this, and there needs to be to statically guarantee resource safety. For starters, we could have the rules:

- **Don't create structures that allow circular ownership dependencies.**
- **Ownership relations must be DAGs**
- **Relationships that make ownerships a DAG (and not just a tree) must be shared_ptrs**

5.6. trusted code

Ideally, we would have no "trusted"/unverified code and have every construct validated. Unfortunately, this seems infeasible. Access to hardware can be impossible to check and some fundamental data structures can be very hard to check (statically or dynamically). Witness that every language offering guarantees offers a loophole, often a way of invoking unchecked C or C++ code.

A "trusted" code option is often overused, so every such should trigger a warning leading to an extra-thorough code review.

We assume that some “trusted”/unverified annotation applying to a block is inevitable. There is currently no such feature in the Core Guidelines, but then the Core Guidelines prevents linking to unchecked code only through an analysis option and it is generally desirable to have the code itself indicate what’s going on. See §7.

6. Run-time checks

Some violations cannot be detected until run time. Examples are access through a **nullptr**, a range error subscripting through a pointer, a range error subscripting a vector, and a range error arising from a wrongly specified range for an algorithm.

Such violations cannot be left as undefined or unspecified behavior while claiming any form of safety. Instead, they must lead to either an exception being thrown or termination (in applications that allow termination).

6.1 Pointers

The C++ Core guidelines library (the GSL) offers a type that checks for **nullptr** at initialization:

```
void f(not_null<int*> p)
{
    // no need to check for nullptr here
}

void g(int* q)
{
    f(q); // run-time check: potential run-time error
}
```

Simple cases of passing a **nullptr** can be caught by the static analyzer, but not all. An alternative design would leave all **nullptr** checks to the static analyzer typedef like **owner** or **[[not_null]]** annotation.

6.2. Subscripting

The **nullptr** check, initialization guarantees, and invalidation checks together guarantee that a pointer is valid (i.e., points to an object). However, subscripting a pointer is inherently unsafe. In general, we don’t reliably know how many elements a pointer points to.

Consequently, the Core Guidelines prevents subscripting of naked pointers and recommends use of abstractions, such as **vector** and **span**. However, the standard doesn’t guarantee range checking of such abstractions with subscripting. For type safety, enforced range checking is essential.

There are several ways of ensuring range checking

- Annotate a type or a use of a type with a **[[check(range)]]** annotation.
- Have a compiler option that adds range checks for all such abstractions.
- Have an in-code annotation, e.g., **[[check(type_safety)]]**, that guarantees that such checks are added

See §7.

6.3. Iterators and Ranges

Many standard algorithms return either an iterator to an element or the **end()**. An **end()** may not be dereferenced:

```
auto p = find(v.begin(),v.end(),42);
if (p!=v.end()) {
    // use *p here
}
else {
    // never use *p here
}
```

There is no way of knowing whether a call returns **end()**, so unless we know more, an iterator returned from an algorithm must be tested against an appropriate **end()** before an attempt to dereference. For general code, no static guarantee of that is possible. Furthermore, unlike for **nullptr** (§6.1), there is no single test that can be used to determine whether an iterator is **end()**. We need an enforceable safe style of use. This is made more complicated because getting back an **end()** is not (necessarily) an error, but just a specific result that must be handled appropriately.

We need something like **not_null** for iterators. A predicate for that must be specific for a container, e.g., **not_end(container,iterator)** that compares **iterator** to **container.end()**. For example

```
auto p = find(v.begin(),v.end(),42);
if (not_end(v,p)) {
    // use *p here
}
else {
    // never use *p here
}
```

The static analyzer would “know about” the **not_end()** predicate and flag dereferences of a returned iterator that had not been checked.

```
auto p = find(v.begin(),v.end(),42);
*p = 7; // bad: not_end(p) missing
```

7. Controls

How and where does a developer control which guarantees should be enforced? There are obvious alternatives:

- Options in the tool chain
- Annotations in the code itself
- Combinations of the two

The C++ core guidelines use controls in the tool chain to avoid touching the compiler and other source code tools. That’s very flexible; a user can choose from dozens of checks that can be individually specified in addition to “profiles” that as collections of actions of related rules and checks (e.g., all rules needed for memory safety). However, this has two problems

- The diversity of options can be hard to handle
- You cannot tell what's intended by looking at the code.

The former can be handled by build-system policies and has the significant advantage that different builds can enforce different sets of rules. This is essential for libraries that need to run in different environments and especially for the essential gradual adoption. The latter is a significant problem.

This “Controls” section builds on the experiences from CGs use, but has not been implemented. The problems of combining new and old code and of combining code with different requirements for guarantees naturally emerge whichever design alternative is used. The CG approach gives us the opportunity to be precise about what is done.

Consider a `[[check(type_safety)]]` annotation placed first in a module or at the top of a TU. To provide a solid guarantee, all calls to this module would have to be from other modules also designated type-safe or pointer arguments might be invalid. Similarly, every module called with pointer arguments from this module must also be designated type-safe or the called module might violate essential guarantees (e.g., by deleting a pointer it didn't own or making a range error). Having all code of a program offer the same guarantees is ideal but doesn't allow for gradual adoption.

We need to consider different kinds of code

- Old – not yet improved – code
- Code that has been – or is being – verified to offer a set of guarantees
- Code that has been deemed infeasible or unsuitable to be verified and must be trusted to be correct (after extensive review and testing)
- Code that has been – or is being – verified to offer a different set of guarantees

For each case, we need to consider the granularity of the code in question:

- A block of code embedded in a single compilation unit
- A separate library (possibly a header-only library)

A developer can exercise or request enforcement of rules enunciated by a given profile (a set of guarantees) along three dimensions:

- Code directly written in a given source file
- Code depended on (e.g., via import)
- Code provided to others via interfaces (e.g., export)

We will first discuss the composition of code using modules and later (7.5) extend that to traditional translation units.

Placing enforcement annotations in the code makes the writer's assumptions explicit and also allows for the compiler and other tools to verify combinations of modules.

7.1. Syntax

We use the annotation syntax, e.g., `[[check(type_safety)]]` because we must have an open set of profiles (e.g., named set of rules/guarantees). Some of them are directly known to the standard; others defined

external tools and organizations. We cannot have an open set of keywords to name them. This notation doesn't imply that the implementation of the set of standard guarantees is optional.

Profiles not listed in the standard cannot be assumed to be known everywhere, and will therefore not be enforced where they are not known. However, since a profile does not introduce non-standard C++ semantics, static properties verified in one place will also hold in places where they couldn't be checked.

7.2. Module guarantees

To ensure that the code in a module does not violate a guarantee, simply state the desired guarantee (or guarantees) as the first item in the module:

```
[[check(type_safety)]];           // activate type_safety profile

int64_t make_pun(double x)
{
    union {
        double d;
        int64_t i;
    } y { x };
    return y.i;           // error: type punning not allowed: use a bit_cast
}

int* glob(int i)
{
    return new int{i};    // error: resource leak
}

// ...
```

Here, the **type_safety** profile is requested for checking all the code in that translation unit. It enables the compiler to reject – at compile time – attempts to use a union.

7.3. Code Fragment guarantees

When we want to enforce a guarantee or leave a guarantee unverified in a piece of code smaller than a module, we can do so by annotating a declaration. For example:

```
[[check(type_safety)]];           // activate type_safety profile

template<regular Elem>
[[suppress(type_safety)]] class Slist {
    // Slist is implemented using unsafe pointer manipulation
    void insert(const Elem&);
    void remove(Elem*);
    // ...
};
```

This kind of unverified code fragment is our variant of the “trusted code” idea. Such code is best avoided. An organization that is serious about guarantees will put processes in place to ensure that unverified code

receives extra scrutiny. The danger is that such suppression will be used especially for complex code that is most error prone. This suppression mechanism may best be left out of an initial implementation even though unverified code occurs naturally in current code and developers are used to rely on it.

Increasing the amount of checking for a piece of code can be useful and reasonable. For example:

```
[[check(arithmetic)]] double compute(double x, double y)
{
    // code here is checked for overflow
}
```

7.4. Using an imported module

Most C++ programs depend on separately developed libraries. The developer of a program component needs the ability to request that uses of facilities from dependencies (libraries) satisfy expectations in effect where the dependency is established.

First, we give a few key examples and then §7.5 gives the general rule.

7.4.1. Use from an unverified module

How do we allow for an unverified module to use a type-safe one? Note that the type-safety guarantees depend on a user obeying the type and resource rule. For example:

```
import std;    // assume that std is type-safe

Std::string s {"asdf"};

void bad()
{
    char* p = &s[0];
    s = "";    // invalidates p
    auto n = strlen(p);
}

```

Naturally, real-world violations will be more subtle.

It should be possible to optionally and automatically add tests (e.g., `nullptr` tests) for cases where a type-safe module is called from a non-safe one. Unfortunately, that forces the writer to try to write run-time tests (probably contracts) for all the guarantees offered by `[[check(type_safety)]]`. For example, given

```
import std [[enable(memory_safety)]];

void f(vector<int>& v)
{
    int& a = v[23];    // the bound is checked
    v.resize(42);    // potential error: dangling reference a
}

```

The first line instructs the compiler to:

- make all facilities from the standard library module **std** available to the current source file
- enforce all uses of the standard library in a manner consistent with the **memory_safety** profile.

Bound safety is enforced (run-time) and the invalidation is detected (compile-time).

7.4.2. Use of an unverified module

How do we allow for a type-safe module to use one that isn't? This option is popular with people who believe in unlimited "trust the programmer" or favor run-time contracts that test assumptions but can be turned off when ultimate performance is needed. For example

```
[[ check(type_safety)]]

import old;    // an unverified module

void use(std::vector<int>& v)
{
  if (v.size()) {
    auto p = v.begin()+v.size()/2;
    old_algo(v);

    *p = 9;
  }
}
```

What if **old_algo()** reallocated the elements of **v** so that **p** became invalidated?

Clearly, this call of **old_algo()** might violate the required **type_safety** guarantee and must by default be an error. On the other hand, because we cannot re-write all code immediately, for the first many years of verified type safety, there must be a way for such code to be made to work. For example:

```
import old [[suppress(type_safety)]];
```

7.5. Export guarantees

A library author can provide guarantees about their components to their users along specific profiles. Consider, for instance, a module providing the type **DynArray** for non-resizable dynamic arrays, with bound checking:

```
export module DataType.Array [[provide(memory_safety)]];

export template<typename T>
struct DynArray {
  // ...
  size_t size() const;
  const T& operator[](size_t i) const
  {
    if (i >= size()) throw OutOfBound{i};
    return data()[i];
  }
};
```

Here the module **DataType.Array** provides the class template **DynArray** along with guarantees are assured by the **memory_safety** profile. All consumers (e.g., direct or indirect importers) of **DataType.Array** are guaranteed – whether they request it or not – that any entity (class or function) that they use from that module provide the guarantees of **memory_safety**. If any such guarantee requires a runtime check, there is no way of turning that off on the import side. That is, once explicitly provided, a profile guarantee cannot be turned off. Of course, during the compilation of the module **DataType.Array**, the rules of the profile **memory_array** are in effect and must be obeyed within the boundaries of the module.

Linking expectations to guarantees

How does the interface of a module **M** meet the guarantees expected at the points of use? The expectation satisfaction is decided purely based on the profiles listed at both the point of declaration (export) and at the point of use (import). Let's say the module **M**'s interface was declared as

```
export module M [[provide(G1, ..., Gn)]];
```

where **G1, ..., Gn** is the list of *n* profiles guaranteed by the module **M**. Consider further, that the module **M** is imported through a declaration of the form

```
import M [[enable(E1, ..., Em)]];
```

where **E1, ..., Em** is the list of *m* profiles expected to be satisfied by declarations exported from **M**. If the set of profiles { **E1, ..., Em** } is a subset of the set of profiles { **G1, ..., Gn** }, then clearly all declarations made available by **M** satisfy the expectations at the points of uses; no further checking is necessary since those profiles are already enforced within the definition of **M**. On the other hand, if at least one profile **Ei** is not in the set { **G1, ..., Gn** }, then further checks are necessary as follows. First of all, the import declaration is *not* an error per se because nothing from it has been used yet. If a declaration *d* attached to **M** is used in the importing source file and that use of *d* (along with its reachable semantic properties) is in a context that fails to satisfy the profile **Ei**, then the compiler issues an error pointing to the usage of *d* failing to meet to expectations of **Ei**.

7.6. Migrating with header files

Existing C++ software is written primarily with the header file/**#include** technology; so how do we propose to apply the discipline/technique outlined above to existing codebases? Well, first of all, to expect having any chances of making significant dent into the safety problem space, we need to start with translation units we can reason with and about. The copy-n-paste technology that unpins **#include** does not lend itself to that sort of analysis, since the tokens from **#included** files are stringed together to form larger token streams with no apparent structures to them until they form a translation unit can be analyzed. However, those header files that can be successfully treated as header units can be imported and the techniques for modules apply equally for header units. As a corollary, starting with identifying and grouping header files into units that can be reasoned about is a necessary step. That is not an unreasonable assumption since software architecture is a basic need that needs to be satisfied with safety issues.

Similarly, the portions of source files called “global module fragments” (that is the token stream in the source file between the first two tokens “module;” and a module declaration) are notionally translation units that can be treated as partitions of the global module that are implicitly imported.

7.7. Standard controls

Without a basic, standard set of profiles, we would be heading for chaos. Here are a few suggestions:

- **type_safety** (no type-or-resource violations)
- **range** (no pointer arithmetic; **span** and **vector** range **throw** or terminate on violations)
- **arithmetic** (no overflow, no narrowing conversions, no implicit signed/unsigned conversions)

8. Acknowledgements

This paper builds directly on the CPP Core guidelines [CG] and the ideas presented in [BS'15].

9. References

- [ISO] ISO C++ standard.
- [CG] [The C++ Core Guidelines](#).
- [GSL] [The Guidelines Support Library](#).
- [BS'15] B. Stroustrup, H. Sutter, and G. Dos Reis: [A brief introduction to C++'s model for type- and resource-safety](#). Isocpp.org. October 2015. Revised December 2015.
- [BS'21] B. Stroustrup: [Type-and-resource safety in modern C++](#). P2410r0. 2021. [CppCon video](#).
- [IPR] G. Dos Reis and B. Stroustrup: [A Principled, Complete, and Efficient Representation of C++](#). Proc. Joint Conference of ASCM 2009 and MACIS 2009.
- [GDR'21] G. Dos Reis: [In-memory and Persistent Representations of C++](#). Video.
- [SELL] B. Stroustrup and G. Dos Reis: [Supporting SELL for High-Performance Computing](#). LCPC 2005.

Appendix: Examples of unsafe C++ and how to avoid them

Here, we present examples of unsafe C++ examples that are not caught by an ordinary C++ compiler and point out how a static analyzer based on an augmented CG would catch them and what alternatives should be available.

A.1. Pointers

Unconstrained use of pointers can be used to break every rule:

```
auto f(int* p, int n)
{
    return p[n];
}
```

Here, **p** might be the **nullptr** and **n** may cause **p[n]** to point outside the array pointed to by **p**. Thus, the CG bans such subscripting when range checking is requested.

```
auto f(span<int> p, int n)
{
    return p[n];
}
```

Here, the subscripting is checked and leads to an exception throw or termination if out of range. For this to work, **span** subscripting must run-time checked (as **gsl::span** is).

A.2. Range errors

???

A.3. Nullptr dereference

???

A.4. Overflow

???

A.5. Data races

???